

# Oh-RAM! One and a Half Round Atomic Memory

Theophanis Hadjistasi <sup>\*</sup>    Nicolas Nicolaou <sup>†</sup>    Alexander Schwarzmann<sup>\*</sup>

October 27, 2016

## Abstract

Emulating atomic read/write shared objects in a message-passing system is a fundamental problem in distributed computing. Considering that network communication is the most expensive resource, efficiency is measured first of all in terms of the communication needed to implement read and write operations. It is well known that two communication round-trip phases involving in total four message exchanges are sufficient to implement atomic operations. It is also known that under certain constraints on the number of readers with respect to the numbers of replica servers and failures it is possible to implement single-writer atomic objects such that each operation involves one round-trip phase, or two message exchanges.

In this work we present a comprehensive treatment of the question on when and how it is possible to implement atomic memory where read and write operations complete in three message exchanges, i.e., we aim for *One and half Round Atomic Memory*, hence the name Oh-RAM! We present algorithms that allow operations to complete in three communication exchanges without imposing any constraints on the number of readers and writers. Specifically, we present an atomic memory implementation for the *single-writer/multiple-reader* (SWMR) setting, where reads complete in *three* communication exchanges and writes complete in *two* exchanges. We pose the question of whether it is possible to implement *multiple-writer/multiple-reader* (MWMR) memory where operations complete in at most three communication exchanges. We answer this question in the negative by showing that an atomic memory implementation is impossible if both read and write operations take *three* communication exchanges. Motivated by this impossibility result, we provide a MWMR atomic memory implementation where reads involve *three* and writes involve *four* communication exchanges. In light of our impossibility result these algorithms are optimal in terms of the number of communication exchanges. We rigorously reason about the correctness of the algorithms.

**Contact Author:** Theophanis Hadjistasi, [theo@uconn.edu](mailto:theo@uconn.edu).

---

<sup>\*</sup>University of Connecticut, Storrs CT, USA. Email: [theo@uconn.edu](mailto:theo@uconn.edu), [aas@engr.uconn.edu](mailto:aas@engr.uconn.edu)

<sup>†</sup>IMDEA Networks Institute, Madrid, Spain. Email: [nicolas.nicolaou@imdea.org](mailto:nicolas.nicolaou@imdea.org)

# 1 Introduction

Emulating atomic [8] (or linearizable [7]) read/write objects in message-passing environments is one of the fundamental problems in distributed computing. Atomicity is the most intuitive consistency semantic as it provides the illusion of a single-copy object that serializes all accesses such that each read operation returns the value of the latest preceding write operation. Solutions to this problem are complicated when the processors participating in implementing the service may fail and when the environment is asynchronous. To cope with processor failures, distributed object implementations like [1] use *redundancy* by replicating the object at multiple network locations (replica servers). Replication introduces the problem of consistency due to the fact that read and write operations may access different object replicas, some of which may contain obsolete object values.

The seminal work of Attiya, Bar-Noy, and Dolev [1] provided an algorithm, colloquially referred to as ABD, that implements single-writer/multiple-reader (SWMR) atomic objects in message-passing crash-prone asynchronous environments. The ordering of operations is accomplished with the help of logical *timestamps* associated with each value. Here each operation is guaranteed to terminate as long as some majority of replica servers do not crash. Each write operation takes one communication round-trip phase, or round, involving *two* message exchanges and each read operation takes two rounds involving in total *four* message exchanges. Showing atomicity of the implementation relies on the fact that any two majorities have a non-empty intersection. Subsequently, [10] showed how to implement multi-writer/multiple-reader (MWMR) atomic memory where both read and write operations involve two communication round trips involving in total four message exchanges.

The work by Dutta et al. [2] introduced a SWMR implementation where both reads and writes involve a single round consisting of *two* communication exchanges. Such an implementation is referred to as *fast*, and it was shown that this is possible only when the number of readers  $r$  is bounded with respect to the number of servers  $s$  and the server failures  $f$ , specifically by  $r < \frac{s}{f} - 2$ , and when there is only a single writer in the system. An interesting observation made in [2] is that atomic memory may be implemented (using a max/min technique) so that each read and write operation complete in *three* communication exchanges. The authors however did not elaborate further on the inherent limitations that such a technique may impose on the distributed system.

Several subsequent works, e.g., [4, 5, 6], focused in relaxing the bound on the number of readers and writers in the service by proposing hybrid approaches where some operations complete in *one* and others in *two* rounds. In addition, [4] provides tight bounds on the number of rounds that *read* and *write* operations require in the MWMR model.

**Contributions.** We address the gap between one-round and two-round implementations by examining the possibility of implementing atomic memory where read and write operations take “one and a half rounds,” i.e., where operations complete in *three* message exchanges. For the asynchronous message-passing environments with process crashes we first present a SWMR algorithm where read operations take *three* and write operations take *two* communication exchanges. Considering the MWMR setting, we show that it is impossible to guarantee atomicity when both read and write operations complete in *three* communication exchanges. We then present a MWMR implementation, where reads take *three* and writes take *four* communication exchanges. Both of our algorithms are *optimal* in terms of communication exchanges for settings without constraints on the number of participants. We rigorously reason about the correctness of the algorithms. Additional details are as follows.

1. We present a new SWMR algorithm for atomic objects in the asynchronous message-passing model with processor crashes. The write operation takes *two* communication exchanges and it is similar to the write operation of ABD. Read operations take *three* communication exchanges: (1) the reader sends a message to servers, (2) the servers share this information, and (3) once this is “sufficiently” done, servers reply to the reader. A key idea of the algorithm is that the reader returns the value that is associated with the *minimum* timestamp (cf. the observation in [2]). The read operations in this algorithm are optimal in terms of communication exchanges. (Section 3.)

Model	Algorithm	Read Exchanges	Write Exchanges	Read Comm.	Write Comm.
SWMR	ABD [1]	4	2	$4 \mathcal{S} $	$2 \mathcal{S} $
SWMR	Oh-SAM	3	2	$ \mathcal{S} ^2 + 2 \mathcal{S} $	$2 \mathcal{S} $
MWMR	ABD [1]	4	4	$4 \mathcal{S} $	$4 \mathcal{S} $
MWMR	Oh-MAM	3	4	$ \mathcal{S} ^2 + 2 \mathcal{S} $	$4 \mathcal{S} $

Table 1: Summary of the complexities.

2. A major part of our contribution is to show the impossibility of MWMR implementations where both write and the read operations take *three* communication exchanges. Specifically, we show that atomicity is violated even in a system that consists of two readers, two writers, and is subject to a single server failure. (Section 4.)
3. Motivated by the impossibility result, we revise the SWMR algorithm to yield a MWMR algorithm. The existence of multiple writers complicates the write operations, and in the new algorithm writes take *four* communication exchanges (cf. [10]). Read operations complete again in *three* communication exchanges. (Section 5.)

The summary of the complexity results and the comparison with ABD [1] is in Table 1. Improvements in latency are obtained in a trade-off with communication complexity. We note that even though the message complexity of communication among servers is higher, in certain practical settings, e.g., data centers, servers are interconnected using low-latency, high-bandwidth communication links. Lastly, our results suggest that definitions of operation “fastness” (cf. [2, 5]) need to reflect the relevant lower bounds without imposing restrictions on the number of readers in multi-reader settings and the number of writers in multi-writer settings.

## 2 Models and Definitions

The system consists of a collection of crash-prone, asynchronous processors with unique identifiers from a totally-ordered set  $\mathcal{I}$  partitioned into: set  $\mathcal{W}$  of writer identifiers, set  $\mathcal{R}$  of reader identifiers, and set  $\mathcal{S}$  of replica server identifiers with each *server* maintaining a copy of the object. Any subset of writers and readers, and up to  $f$  servers,  $f < \frac{|\mathcal{S}|}{2}$ , may crash at any time. Processors communicate by exchanging messages via asynchronous point-to-point reliable channels; messages may be reordered. For convenience we use the term *broadcast* as a shorthand denoting sending point-to-point messages to multiple destinations.

**Executions.** An algorithm  $A$  is a collection of processes, where process  $A_p$  is assigned to processor  $p \in \mathcal{I}$ . The *state* of processor  $p$  is determined over a set of state variables, and the state of  $A$  is a vector that contains the state of each individual process. Algorithm  $A$  performs a *step*, when some process  $p$  atomically (i) receives a message, (ii) performs local computation, (iii) sends a message. Each such action causes the state at  $p$  to change from pre-state  $\sigma_p$  to a post-state  $\sigma'_p$ . Also, the state of  $A$  changes from  $\sigma$  to  $\sigma'$  where  $\sigma$  contains state  $\sigma_p$  for  $p$  and  $\sigma'$  contains state  $\sigma'_p$ . An *execution fragment* is an alternating sequence of states and actions of  $A$  ending in a state. An *execution* is an execution fragment that starts with the initial state. We say that an execution fragment  $\xi'$  extends an execution fragment  $\xi$  if the last state of  $\xi$  is the first state of  $\xi'$ . A process  $p$  *crashes* in an execution if it stops taking steps; otherwise  $p$  is *correct*.

**Atomicity.** An implementation of a read or a write operation contains an *invocation* action (such as a call to a procedure) and a *response* action (such as a return from the procedure). An operation  $\pi$  is *complete* in an execution  $\xi$ , if  $\xi$  contains both the invocation and the *matching* response actions for  $\pi$ ; otherwise  $\pi$  is *incomplete*. An execution is *well formed* if any process invokes one operation at a time. Finally we say that an operation  $\pi$  *precedes* an operation  $\pi'$  in an execution  $\xi$ , denoted by  $\pi \rightarrow \pi'$ , if

the response step of  $\pi$  appears before the invocation step in  $\pi'$  in  $\xi$ . Two operations are *concurrent* if neither precedes the other. The correctness of an atomic read/write object implementation is defined in terms of the *atomicity* (safety) and *termination* (liveness) properties. Termination requires that any operation invoked by a correct process eventually completes. The atomicity property is defined following [9]. For any execution  $\xi$ , if all invoked read and write operations are complete, then the operations can be partially ordered by an ordering  $\prec$ , so that the following properties are satisfied:

- P1** The partial order  $\prec$  is consistent with the external order of invocation and responses, that is, there do not exist operations  $\pi$  and  $\pi'$ , such that  $\pi$  completes before  $\pi'$  starts, yet  $\pi' \prec \pi$ .
- P2** All write operations are totally ordered and every read operation is ordered with respect to all writes.
- P3** Every read operation returns the value of the last write preceding it in the partial order, and any read operation ordered before all writes returns the initial value of the object.

**Efficiency Metrics and Message Exchanges.** Efficiency of implementations is assessed in terms of *message complexity* that measures the worst-case number of messages exchanged during an operation, and *operation latency* that is determined by the *computation time* and the *communication delays*. Computation time accounts for the computation steps that the algorithm performs in each read or write operation. Communication delays are measured in terms of *communication exchanges*, defined next.

The protocol implementing each operation involves a collection of sends (or broadcasts) of typed messages and the corresponding receives. *Communication exchange* within an execution of an operation is the set of sends and receives for the specific message type within the protocol. Note that using this definition, traditional implementations in the style of ABD are structured in terms of *rounds*, cf. [1, 3, 6], where each round consists of two message exchanges, the first, a broadcast, is initiated by the process executing an operation, and the second, a convergecast, consists of responses to the initiator. The number of messages that a process expects during a convergecast depends on the implementation. For example, if a process expects messages from a majority of servers, then  $\frac{|\mathcal{S}|}{2} + 1$  messages are sufficient.

### 3 SWMR Algorithm Oh-SAM

We now present our SWMR algorithm Oh-SAM: *One and a half Round Single-writer Atomic Memory*. The pseudocode is given in Algorithm 1. The write operation takes *two* communication exchanges (similarly to ABD). Read operations take *three* communication exchanges: (1) the reader sends message to servers, (2) each server that receives the request *relays* the request to all servers, and (3) once a server receives the relay for a particular read from a majority of servers, it replies to the reader. The read completes once it collects a majority of these replies. A key idea of the algorithm is that the reader returns the value that is associated with the *minimum* timestamp. Now we give additional details.

Counter variables *read\_op*, *operations* and *relays* are used to help processes identify “new” read and write operations, and distinguish “fresh” from “stale” messages (since messages can be reordered). The value of the object and its associated timestamp, as known by each process, are stored in variables *v* and *ts* respectively. Set *Q*, at each reader  $r_i$ , stores all the received messages. Variable *minTS* holds the minimum timestamp discovered in the received messages.

**Writer Protocol.** Writer  $w$  increments its local timestamp *ts* and broadcasts a *writeRequest* message to servers  $s \in \mathcal{S}$  (lines 23-24). It terminates when at least  $|\mathcal{S}|/2 + 1$  replies are collected (lines 25-26).

**Read Protocol.** Reader  $r$  creates a *readRequest*, with its id  $r$  and its local operation counter *read\_op*, and broadcasts this message to servers  $s \in \mathcal{S}$  (line 10). It then waits to collect at least  $|\mathcal{S}|/2 + 1$  messages from servers. When “fresh” messages are collected from a majority of servers, the reader returns the value  $v$  associated with the *minimum* *ts* among the received messages (lines 14-16).

---

**Algorithm 1** Reader, Writer, and Server Protocols for SWMR algorithm Oh-SAM

---

```

1: At each reader  $r_i$ 
2: Variables:
3:  $ts \in \mathbb{N}^+$ ,  $minTS \in \mathbb{N}^+$ ,  $v \in V$ 
4:  $read\_op \in \mathbb{N}^+$ ,  $Q \subseteq S \times M$ 
5: Initialization:
6:  $ts \leftarrow 0$ ,  $minTS \leftarrow 0$ ,  $read\_op \leftarrow 0$ 
7:  $v \leftarrow \perp$ ,  $Q \leftarrow \emptyset$ 
8: function READ
9:    $read\_op \leftarrow read\_op + 1$ 
10:  broadcast ( $\langle readRequest, r_i, read\_op \rangle$ ) to  $S$ 
11:  wait until  $|\mathcal{S}|/2 + 1$  server messages  $m$ 
12:    with  $(m.read\_op = read\_op)$ 
13:    Let  $Q = \{(s, m) | r_i \text{ received } m \text{ from } s\}$ 
14:     $minTS \leftarrow \min\{m.ts' | m \in Q\}$ 
15:     $v = m.val$  such that  $m \in Q \wedge m.ts' = minTS$ 
16:  return( $v$ )

17: At writer  $w$ 
18: Variables:
19:  $ts \in \mathbb{N}^+$ ,  $v \in V$ 
20: Initialization:
21:  $ts \leftarrow 0$ ,  $v \leftarrow \perp$ 
22: function WRITE( $val : input$ )
23:    $\langle ts, v \rangle \leftarrow \langle ts + 1, val \rangle$ 
24:   broadcast ( $\langle writeRequest, ts, v, w \rangle$ )  $\forall s \in \mathcal{S}$ 
25:   wait until  $|\mathcal{S}|/2 + 1$  writeAck messages  $m$ 
26:     with  $(m.ts = ts)$ 
27:   return

28: At server  $s_i$ 
29: Variables:
30:  $ts \in \mathbb{N}^+$ ,  $v \in V$ 
31:  $operations[1...|\mathcal{R}| + 1]$ ,  $relays[1...|\mathcal{R}| + 1]$ 
32: Initialization:
33:  $ts \leftarrow 0$ ,  $v \leftarrow \perp$ 
34:  $operations[i] \leftarrow 0$  for  $i \in \mathcal{R}$ ,  $relays[i] \leftarrow 0$  for  $i \in \mathcal{R}$ 

35: Upon receive( $\langle readRequest, r_i, read\_op \rangle$ )
36:   broadcast ( $\langle readRelay, ts, v, r_i, read\_op, s_i \rangle$ ) to  $S$ 

37: Upon receive( $\langle writeRequest, ts', v', w \rangle$ )
38: if ( $ts < ts'$ ) then
39:    $\langle ts, v \rangle \leftarrow \langle ts', v' \rangle$ 
40: send ( $\langle writeAck, ts, v, s_i \rangle$ ) to  $w$ 

41: Upon receive( $\langle readRelay, ts', v', r_i, read\_op, s_i \rangle$ )
42: if ( $ts < ts'$ ) then
43:    $\langle ts, v \rangle \leftarrow \langle ts', v' \rangle$ 
44: if ( $operations[r_i] < read\_op$ ) then
45:    $operations[r_i] \leftarrow read\_op$ 
46:    $relays[r_i] \leftarrow 0$ 
47: if ( $operations[r_i] = read\_op$ ) then
48:    $relays[r_i] \leftarrow relays[r_i] + 1$ 
49: if ( $relays[r_i] = |\mathcal{S}|/2 + 1$ ) then
50:   send ( $\langle readAck, ts, v, read\_op, s_i \rangle$ ) to  $r_i$ 

```

---

**Server Protocol.** (1) Upon receiving message  $\langle readRequest, r_i, read\_op \rangle$ , the server creates a *readRelay* message, with its pair  $ts, v$  and its id  $s_i$  and broadcasts it to all servers  $s \in \mathcal{S}$  (lines 35-36).

(2) Upon receiving message  $\langle readRelay, ts', v', r_i, read\_op \rangle$  server  $s$  compares its local timestamp  $ts$  with  $ts'$  enclosed in the message. If  $ts < ts'$ , then  $s$  sets local timestamp-value pair to the ones enclosed in the message, i.e.  $\langle ts, v \rangle = \langle ts', v' \rangle$  (lines 42-43). Next,  $s$  checks if the received *readRelay* marks a new read operation by  $r_i$ , i.e.,  $read\_op > operations[r_i]$  (line 44). If this holds, then  $s$ : a) sets its local counter for  $r_i$  to the received counter, i.e.,  $operations[r_i] = read\_op$ ; and b) re-initializes the relay counter for  $r_i$  to zero, i.e.,  $relays[r_i] = 0$  (lines 44-46). Server  $s$  also updates the number of collected *readRelay* messages regarding the read request created by reader  $r_i$  (lines 47-48). When  $s$  receives  $\langle readRelay, ts, v, read\_op, s_i \rangle$  from a majority of servers, it creates a  $\langle readAck, ts, v, read\_op, s_i \rangle$  message and sends it to reader  $r_i$  (lines 49-50).

(3) Upon receiving message  $\langle writeRequest, ts', v', w \rangle$  server  $s$  compares its local timestamp  $ts$  with the received one,  $ts'$ . If  $ts < ts'$ , then the server updates its local timestamp and local value to be equal to the ones in the received message  $\langle ts, v \rangle = \langle ts', v' \rangle$  (lines 38-39). In any other case, no updates are taking place. Finally, the server sends an acknowledgement message to the requesting writer.

## 4 Correctness of SWMR Algorithm Oh-SAM

To prove correctness of algorithm Oh-SAM (Algorithm 1) we show that the *liveness* (termination) and *atomicity* (safety) properties are satisfied. The termination is satisfied with respect to our failure model: up to  $f$  servers may fail, where  $f < |\mathcal{S}|/2$  and each operation waits for messages from a majority of servers,  $|\mathcal{S}|/2 + 1$ . We now present the proof, with selected details given in the appendix.

**Atomicity.** To prove the atomicity properties we order operations by means of the timestamps used by each operation, expressing the required (to be proved) partial order as follows.

- A1** If a *read*  $\rho$  succeeds a *write*  $\omega$ , where  $\omega$  writes value with timestamp  $ts$  and  $\rho$  returns the value for timestamp  $ts'$ , then  $ts' \geq ts$ .
- A2** If a write operation  $\omega_1$  that writes the value with timestamp  $ts_1$  precedes a write operation  $\omega_2$  that writes the value with timestamp  $ts_2$ , i.e.,  $\omega_1 \rightarrow \omega_2$ , then  $ts_2 > ts_1$ .
- A3** If  $\rho_1$  and  $\rho_2$  are two read operations such that  $\rho_1 \rightarrow \rho_2$  and  $\rho_1$  returns the value with timestamp  $ts_1$  and  $\rho_2$  returns the value with timestamp  $ts_2$ , then  $\rho_2$  returns  $ts_2 \geq ts_1$ .



Property A2 follows from *well-formedness* of the sole writer in the system and the fact that the writer always increments the timestamp. For paucity of space we present the complete proofs for lemmas 1 and 2 in the optional Appendix. It is easy to see that the  $ts$  variable in each server  $s$  is monotonically increasing. This leads to the following lemmas.

**Lemma 1** *In any execution  $\xi$  of the algorithm, if a server  $s$  replies with a timestamp  $ts$  at time  $T$ , then server  $s$  replies with a timestamp  $ts' \geq ts$  at any time  $T' > T$ .*

**Lemma 2** *In any execution  $\xi$  of the algorithm, if a server  $s$  receives a timestamp  $ts$  from process  $p$ , then  $s$  attaches a timestamp  $ts' \geq ts$  in any subsequent message it sends.*

As a next step we show how atomicity Property A3 is satisfied.

**Lemma 3 (Property A3)** *In any execution  $\xi$  of the algorithm, if  $\rho_1$  and  $\rho_2$  are two read operations such that  $\rho_1$  precedes  $\rho_2$ , i.e.,  $\rho_1 \rightarrow \rho_2$ , and  $\rho_1$  returns the value for timestamp  $ts_1$ , then  $\rho_2$  returns the value for timestamp  $ts_2 \geq ts_1$ .*

**Proof.** Let the two operations  $\rho_1$  and  $\rho_2$  be invoked by processes with identifiers  $r_1$  and  $r_2$  respectively (not necessarily different). Also, let  $RSet_1$  and  $RSet_2$  be the sets of servers that sent a *readAck* message to  $r_1$  and  $r_2$  during  $\rho_1$  and  $\rho_2$ .

Assume by contradiction that read operations  $\rho_1$  and  $\rho_2$  exist such that  $\rho_2$  succeeds  $\rho_1$ , i.e.,  $\rho_1 \rightarrow \rho_2$ , and the operation  $\rho_2$  returns a timestamp  $ts_2$  that is smaller than the  $ts_1$  returned by  $\rho_1$ , i.e.,  $ts_2 < ts_1$ . According to our algorithm,  $\rho_2$  returns a timestamp  $ts_2$  that is smaller than the minimum timestamp received by  $\rho_1$ , i.e.,  $ts_1$ , if  $\rho_2$  obtains  $ts_2$  and  $v$  in the *readAck* message of some server  $s_x \in RSet_2$ , and  $ts_2$  is the minimum timestamp received by  $\rho_2$ .

Let us examine if  $s_x$  replies with  $ts'$  and  $v'$  to  $\rho_1$ , i.e.,  $s_x \in RSet_1$ . By Lemma 1, and since  $\rho_1 \rightarrow \rho_2$ , then it must be the case that  $ts' \leq ts_2$ . According to our assumption  $ts_1 > ts_2$ , and since  $ts_1$  is the smallest timestamp sent to  $\rho_1$  by any server in  $RSet_1$ , then it follows that  $r_1$  does not receive the *readAck* message from  $s_x$ , and hence  $s_x \notin RSet_1$ .

Now let us examine the actions of the server  $s_x$ . From the algorithm, server  $s_x$  collects *readRelay* messages from a majority of servers in  $\mathcal{S}$  before sending a *readAck* message to  $\rho_2$  (lines 49 - 50). Let  $RRSet_{s_x}$  denote the set of servers that sent *readRelay* to  $s_x$ . Since, both  $RRSet_{s_x}$  and  $RSet_1$  contain some majority of the servers then it follows that  $RRSet_{s_x} \cap RSet_1 \neq \emptyset$ .

This means that there exists a server  $s_i \in RRSet_{s_x} \cap RSet_1$ , which sent (i) a *readAck* message to  $r_1$  for  $\rho_1$ , and (ii) a *readRelay* message to  $s_x$  during  $\rho_2$ . Note that  $s_i$  sends a *readRelay* for  $\rho_2$  only after it receives a read request from  $\rho_2$  (lines 35-36). Since  $\rho_1 \rightarrow \rho_2$ , then it follows that  $s_i$  sent the *readAck* to  $\rho_1$  before sending the *readRelay* to  $s_x$ . Thus, by Lemma 2, if  $s_i$  attaches a timestamp  $ts_{s_i}$  in the *readAck* to  $\rho_1$ , then  $s_i$  attaches a timestamp  $ts'_{s_i}$  in the *readRelay* message to  $s_x$ , such that  $ts'_{s_i} \geq ts_{s_i}$ . Since  $ts_1$  is the minimum timestamp received by  $\rho_1$ , then  $ts_{s_i} \geq ts_1$ , and hence  $ts'_{s_i} \geq ts_1$  as well. By Lemma 2, and since  $s_x$  receives the *readRelay* message from  $s_i$  before sending a *readAck* to  $\rho_2$ , it follows that  $s_x$  sends a timestamp  $ts_2 \geq ts'_{s_i}$ . Therefore,  $ts_2 \geq ts_1$  and this contradicts our initial assumption.  $\square$

We now show that any read operation following a write operation receives *readAck* messages from the servers where each included timestamp is at least the one returned by the complete write operation.

**Lemma 4** *In any execution  $\xi$  of the algorithm, if a read operation  $\rho$  succeeds a write operation  $\omega$  that writes  $ts$  and  $v$ , i.e.,  $\omega \rightarrow \rho$ , and receives *readAck* messages from a majority of servers  $RSet$ , then each  $s \in RSet$  sends a *readAck* message to  $\rho$  with a timestamp  $ts_s$  s.t.  $ts_s \geq ts$ .*

**Proof.** Suppose that  $WSet$  is the set of servers that send a *writeAck* message to the write operation  $\omega$  and let  $RRSet$  denote the set of servers that sent *readRelay* messages to server  $s$ .

According to lemma assumption, write operation  $\omega$  is completed. By Lemma 2, we know that if a server  $s$  receives a timestamp  $ts$  from a process  $p$  at some time  $T$ , then  $s$  attaches a timestamp  $ts'$  s.t.  $ts' \geq ts$  in any message sent at any time  $T' \geq T$ . This, means that every server in  $WSet$ , sent a *writeAck* message to  $\omega$  with a timestamp greater or equal to  $ts$ . Hence, every server  $s_x \in WSet$  has a timestamp  $ts_{s_x} \geq ts$ . Let us now examine a timestamp  $ts_s$  that server  $s$  sends to read operation  $\rho$ .

Before server  $s$  sends a *readAck* message to  $\rho$ , it must receive *readRelay* messages from the majority of servers,  $RRSet$  (lines 49 - 50). Since both  $WSet$  and  $RRSet$  contain a majority of servers, then  $WSet \cap RRSet \neq \emptyset$ . By Lemma 2, any server  $s_x \in WSet \cap RRSet$  has a timestamp  $ts_{s_x}$  s.t.  $ts_{s_x} \geq ts$ . Since server  $s_x \in RRSet$  and from the algorithm, server's  $s$  timestamp is always updated to the highest timestamp it noticed (lines 42-43), then when server  $s$  receives the message from  $s_x$ , it will update its timestamp  $ts_s$  s.t.  $ts_s \geq ts_{s_x}$ . Furthermore, server  $s$  creates a *readAck* message for  $\rho$  where it includes its local timestamp and its local value,  $\langle ts_s, v_s \rangle$  (lines 49-50). Each  $s \in RSet$  sends a *readAck* to  $\rho$  with a timestamp  $ts_s$  s.t.  $ts_s \geq ts_{s_x} \geq ts$ . Therefore,  $ts_s \geq ts$  holds and the lemma follows.  $\square$

At this point we want to show that if a read operation succeeds a write operation, then it returns a value at least as recent as the one written.

**Lemma 5 (Property A1)** *In any execution  $\xi$  of the algorithm, if a read  $\rho$  succeeds a write operation  $\omega$  that writes timestamp  $ts$ , i.e.  $\omega \rightarrow \rho$ , and returns a timestamp  $ts'$ , then  $ts' \geq ts$ .*

**Proof.** Let's suppose that read operation  $\rho$  receives *readAck* messages from a majority of servers  $RSet$ . By lines 14 - 16 of the algorithm, it follows that  $\rho$  decides on the minimum timestamp,  $ts' = ts\_min$ , among all the timestamps in the *readAck* messages of the servers in  $RSet$ . From Lemma 4,  $ts\_min \geq ts$  holds, where  $ts$  is the timestamp written by the last complete write operation  $\omega$ , then  $ts' = ts\_min \geq ts$  also holds. Therefore,  $ts' \geq ts$  holds and the lemma follows.  $\square$

And the main result of this section follows.

**Theorem 6** *Algorithm Oh-SAM implements an atomic SWMR register.*

## 5 Impossibility Result

We next show that it is impossible to implement atomic read/write objects in an asynchronous, message-passing system with crash-prone processors where read and write operations take three communication exchanges. We consider algorithms that implement a write operation  $\pi$  invoked by process  $p$  according to the following three-phase scheme: (1) the invoker  $p$  sends a message to a set of servers; (2) each server that receives the message from  $p$  sends a certain relay message to a set of servers; and (3) once a server receives “enough” relay messages it replies to  $p$ . Each phase involves a communication exchange.

To motivate the proof, we briefly explain why it is reasonable to use such a three-phase scheme. First of all, the servers cannot know about a write operation unless the corresponding writer contacts them, thus it must be the writer who initiates phase (1). Moreover, since asynchrony makes it impossible to distinguish slow servers from crashed servers, the writer must include all servers in this phase. In phase (3) it must be the servers who inform the writer about the status/completion of the write operation. Otherwise, either the third phase is unnecessary for the writer, or the writer will wait indefinitely. From the above reasoning, phase (2) must be the transitional phase for the servers to move from phase (1) to phase (3). Hence, phase (2) must facilitate the dissemination of the information regarding any write operation to the rest of the servers.

We use the notation  $rcv(\pi)_{s,p}$  to denote the receipt at server  $s$  of an operation request for an operation  $\pi$  from process  $p$ . Similarly, we denote by  $rcv\_relay(\pi)_{s,s'}$  the receipt at  $s$  of the relay sent by  $s'$  for  $\pi$ .

**Theorem 7** *It is not possible to obtain an atomic MWMR read/write register implementation, where all operations perform three communication exchanges, when  $|\mathcal{W}| = |\mathcal{R}| = 2$ ,  $|\mathcal{S}| \geq 3$  and  $f = 1$ .*

**Proof.** We assume to derive a contradiction that there exists such an implementation  $A$ . Let  $\{w_1, w_2\}$  be the two writers and  $\{r_1, r_2\}$  the two readers. We build a series of executions and we examine if atomicity can be preserved.

Consider an execution  $\xi_1$  that contains two write operations  $\omega_1$  and  $\omega_2$ , performed by  $w_1$  and  $w_2$  respectively, such that  $\omega_1 \rightarrow \omega_2$ . We assume that all servers in  $\mathcal{S}$  receive the messages of the two writes and send relay messages. Some specific server  $s_k$  receives all these relays, while the rest of servers receive the relay messages from all servers, except those from server  $s_k$ . In more detail, for each server  $\forall s_i, s_j \in \mathcal{S} - \{s_k\}$  the receipt of the messages is in the following order: (i)  $rcv(\omega_1)_{s_i, w_1}$ , (ii)  $rcv\_relay(\omega_1)_{s_i, s_j}$ , (iii)  $rcv(\omega_2)_{s_i, w_2}$ , and (iv)  $rcv\_relay(\omega_2)_{s_i, s_j}$ .

W.l.o.g. assume that each server  $s \in \mathcal{S}$  receives the relay message from server  $s_i$  before the message from server  $s_j$  for  $i < j$ . Also notice that, since  $f = 1$ , then each server in  $\mathcal{S} - \{s_k\}$  replies to the write operation without waiting for the relay message from  $s_k$ , since it cannot wait for more than  $|\mathcal{S}| - 1$  relay messages. We now extend  $\xi_1$  with a read  $\rho_1$  from  $r_1$  and we obtain execution  $\xi'_1$ . The messages from  $\rho_1$  reach all servers and all servers relay information regarding  $\rho_1$ . Suppose that the servers receive the relays of all other servers, except for the relay of server  $s_k$ . Again, since  $f = 1$ , the servers cannot wait for the relay from  $s_k$  before replying to  $r_1$ . Server  $s_k$  receives the relays of all servers, including its own relay, and replies to  $r_1$ . In particular, each server  $\forall s_i, s_j \in \mathcal{S} - \{s_k\}$  receives the messages in the following order: (i)  $rcv(\rho_1)_{s_i, r_1}$ , and (ii)  $rcv\_relay(\rho_1)_{s_i, s_j}$ . Server  $s_k$  receives the following messages: (i)  $rcv(\rho_1)_{s_k, r_1}$ , and (ii)  $rcv\_relay(\rho_1)_{s_k, s_j}$ ,  $\forall s_j \in \mathcal{S}$ . As before, assume that all servers receive the relays in the same order. Reader  $r_1$  receives replies from all servers. Since every server receives the same messages in the same order then all servers (including  $s_k$ ) reply to  $r_1$  with the same value, say  $v_{\omega_2}$ . Also, since  $\omega_1 \rightarrow \omega_2 \rightarrow \rho_1$  in  $\xi'_1$ , then  $\rho_1$  must return the value written by  $\omega_2$  in  $\xi'_1$  to preserve atomicity.

Let  $\xi_2$  be similar to  $\xi_1$  but the write operations are performed in the reverse order, i.e.,  $\omega_2 \rightarrow \omega_1$ . In particular, each server  $\forall s_i, s_j \in \mathcal{S} - \{s_k\}$  receives the messages in the following order: (i)  $rcv(\omega_2)_{s_i, w_2}$ , (ii)  $rcv\_relay(\omega_2)_{s_i, s_j}$ , (iii)  $rcv(\omega_1)_{s_i, w_1}$ , and (iv)  $rcv\_relay(\omega_1)_{s_i, s_j}$ . The messages are delivered in the same order as in  $\xi_1$ . Let us now extend  $\xi_2$  by a read operation  $\rho_1$  and obtain  $\xi'_2$ . Moreover, let the messages due to the read operation be delivered in the same manner as in  $\xi'_1$ . In other words: (i) all servers receive the messages from the reader, (ii) all servers receive relays from all other servers except  $s_k$ , and (iii)  $s_k$  receives the relays from all servers. The reader in  $\xi'_2$  receives all replies from the servers. Since, in  $\xi'_2$  the operation order is  $\omega_2 \rightarrow \omega_1 \rightarrow \rho_1$ , then  $\rho_1$  must return the value written by  $\omega_1$  in  $\xi'_2$  to preserve atomicity. Again here notice that the servers reply with the same value, say  $v_{\omega_1}$ .

Consider now an execution  $\xi_3$  where the two write operations are concurrent. In this case there is a set of servers that receive the write request from  $\omega_1$  before the request of  $\omega_2$ , and another set of servers that receive the two requests in the reverse order. In particular we split the set of servers into two sets  $\mathcal{S}_1 \subseteq \mathcal{S} - \{s_k\}$  and  $\mathcal{S}_2 \subseteq \mathcal{S} - \{s_k\}$ , s.t.  $\mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset$ . The following table presents the order in which each server in  $\mathcal{S}_1$  and  $\mathcal{S}_2$  receives the messages:

Message order at $s_i \in \mathcal{S}_1$	Message order at $s_i \in \mathcal{S}_2$
<ul style="list-style-type: none"> <li>• <math>rcv(\omega_1)_{s_i, w_1}</math></li> <li>• <math>rcv\_relay(\omega_1)_{s_i, s_j}</math>, <math>\forall s_j \in \mathcal{S}_1</math></li> <li>• <math>rcv(\omega_2)_{s_i, w_2}</math></li> <li>• <math>rcv\_relay(\omega_2)_{s_i, s_j}</math>, <math>\forall s_j \in \mathcal{S} - \{s_k\}</math></li> <li>• <math>rcv\_relay(\omega_1)_{s_i, s_j}</math>, <math>\forall s_j \in \mathcal{S}_2</math></li> </ul>	<ul style="list-style-type: none"> <li>• <math>rcv(\omega_2)_{s_i, w_2}</math></li> <li>• <math>rcv\_relay(\omega_2)_{s_i, s_j}</math>, <math>\forall s_j \in \mathcal{S}_2</math></li> <li>• <math>rcv(\omega_1)_{s_i, w_1}</math></li> <li>• <math>rcv\_relay(\omega_1)_{s_i, s_j}</math>, <math>\forall s_j \in \mathcal{S} - \{s_k\}</math></li> <li>• <math>rcv\_relay(\omega_2)_{s_i, s_j}</math>, <math>\forall s_j \in \mathcal{S}_1</math></li> </ul>

Let us assume that each server in  $\mathcal{S}_1$  sends a relay for  $\omega_1$  before receiving the request for  $\omega_2$  and each server in  $\mathcal{S}_2$  sends a relay for  $\omega_2$  before receiving the request for  $\omega_1$ . Server  $s_k$  receives the messages as the servers in one of the sets  $\mathcal{S}_1$  and  $\mathcal{S}_2$  and in addition it receives its own relays. We say that  $s_k$  follows *scheme1* if it receives the messages in the same order as any server in  $\mathcal{S}_1$ , or  $s_k$  follows *scheme2* otherwise. The table presents the messages at  $s_k$  in each scheme:



$s_k$ follows <i>scheme1</i>	$s_k$ follows <i>scheme2</i>
<ul style="list-style-type: none"> <li>• <math>rcv(\omega_1)_{s_k, w_1}</math></li> <li>• <math>rcv\_relay(\omega_1)_{s_k, s_j}, \forall s_j \in \mathcal{S}_1 \cup \{s_k\}</math></li> <li>• <math>rcv(\omega_2)_{s_k, w_2}</math></li> <li>• <math>rcv\_relay(\omega_2)_{s_k, s_j}, \forall s_j \in \mathcal{S}</math></li> <li>• <math>rcv\_relay(\omega_1)_{s_i, s_j}, \forall s_j \in \mathcal{S}_2</math></li> </ul>	<ul style="list-style-type: none"> <li>• <math>rcv(\omega_2)_{s_k, w_2}</math></li> <li>• <math>rcv\_relay(\omega_2)_{s_k, s_j}, \forall s_j \in \mathcal{S}_2 \cup \{s_k\}</math></li> <li>• <math>rcv(\omega_1)_{s_k, w_1}</math></li> <li>• <math>rcv\_relay(\omega_1)_{s_k, s_j}, \forall s_j \in \mathcal{S}</math></li> <li>• <math>rcv\_relay(\omega_2)_{s_i, s_j}, \forall s_j \in \mathcal{S}_1</math></li> </ul>

Let  $s_k$  follows *scheme2* in  $\xi_3$ . We extend  $\xi_3$  by a read operation  $\rho_1$  with the same communication as in executions  $\xi'_1$  and  $\xi'_2$ , and we obtain  $\xi'_3$ . Notice that if  $\mathcal{S}_1 = \mathcal{S} - \{s_k\}$ ,  $\mathcal{S}_2 = \emptyset$ , and  $s_k$  follows *scheme1*, then no process can distinguish  $\xi'_3$  from  $\xi'_1$ . Thus, in this case each server replies with  $v_{\omega_2}$  and  $\rho_1$  returns the value written in  $\omega_2$ . If on the other hand  $\mathcal{S}_2 = \mathcal{S} - \{s_k\}$ ,  $\mathcal{S}_1 = \emptyset$ , and  $s_k$  follows *scheme2*, then no process can distinguish  $\xi'_3$  from  $\xi'_2$ . Thus, in this case each server replies with  $v_{\omega_1}$  and  $\rho_1$  returns the value written in  $\omega_1$ .

Let us examine what happens in  $\xi'_3$  when neither  $\mathcal{S}_1$  nor  $\mathcal{S}_2$  is empty. Consider the relays received by a server  $s \in \mathcal{S}_1$ . First,  $s$  receives relays from every server  $s_j \in \mathcal{S}_1$ . Since each server in  $\mathcal{S}_1$  sends a relay for  $\omega_1$  before receiving the request for  $\omega_2$ , then each of those relays do not include any information about  $\omega_2$ . Similarly, each relay received by any server in  $\mathcal{S}_2$  for  $\omega_2$  declares the existence of  $\omega_2$  and not  $\omega_1$ . The relays however received for  $\omega_2$  from each server  $s_j \in \mathcal{S}_1$  declares that  $\omega_1$  happened before  $\omega_2$  and similarly every relay received for  $\omega_1$  from each server  $s_j \in \mathcal{S}_2$  declares that  $\omega_2$  happened before  $\omega_1$ . Also any relay for  $\rho_1$  received by  $s$  from servers in  $\mathcal{S}_1$ , declare that  $\omega_1$  appeared before  $\omega_2$ , and any relay from servers in  $\mathcal{S}_2$  that  $\omega_1$  appeared before  $\omega_2$ . With similar arguments we can show that any relay received at a server  $s \in \mathcal{S}_2$  from a server in  $\mathcal{S}_1$ , declares that  $\omega_1$  happened before  $\omega_2$  and any relay from a server  $\mathcal{S}_2$  declares that  $\omega_2$  happens before  $\omega_1$ .

Let us examine now the relays sent and received in executions  $\xi'_1$  and  $\xi'_2$ . Each server  $s \in \mathcal{S}$  sends a relay for  $\omega_1$  before receiving the write request for  $\omega_2$ . On the other hand,  $s$  sends a relay for  $\omega_2$  after receiving both requests, first from  $\omega_1$  and then from  $\omega_2$ . So every relay of  $\omega_1$  received at a server  $s' \in \mathcal{S}$  bears no information about  $\omega_2$ , and every relay of  $\omega_2$  declares that  $\omega_1$  happened before  $\omega_2$ . A similar situation happens in  $\xi'_2$  with every relay of  $\omega_2$ , not aware of  $\omega_1$ , and every relay of  $\omega_1$  to declare that  $\omega_2$  appeared before  $\omega_1$ . According to our assumptions, each  $s \in \mathcal{S}$  replies with a value  $v_{\omega_2}$  in  $\xi'_1$  and with value  $v_{\omega_1}$  in  $\xi'_2$ .

So *all* relays received in  $\xi'_1$  declare that  $\omega_1$  occurs before  $\omega_2$ , and all relays received in  $\xi'_2$  declare that  $\omega_2$  occurs before  $\omega_1$ . This is also the case in  $\xi'_3$  when either  $\mathcal{S}_1 = \mathcal{S} - \{s_k\}$  or  $\mathcal{S}_2 = \mathcal{S} - \{s_k\}$ . But what would be the value returned by a server  $s$  if some of the relays declare that  $\omega_1$  is before  $\omega_2$ , and some that  $\omega_2$  is before  $\omega_1$ . Since, when  $\mathcal{S}_1 = \mathcal{S} - \{s_k\}$  and  $\mathcal{S}_2 = \emptyset$ ,  $s$  returns  $v_{\omega_2}$ , and, when  $\mathcal{S}_1 = \emptyset$  and  $\mathcal{S}_2 = \mathcal{S} - \{s_k\}$ ,  $s$  returns  $v_{\omega_1}$ , then there should be a point for  $s$  that changes its decision from  $v_{\omega_2}$  to  $v_{\omega_1}$  depending on the number of relays that declare that  $\omega_1$  occurs before  $\omega_2$ , and the number of relays that declare otherwise. In particular, there exists a number  $x$  s.t, if  $s$  witnesses  $x$  relays before declaring that  $\omega_1$  occurs before  $\omega_2$ , and  $|\mathcal{S}| - x$  relays to declare otherwise, then it returns  $v_{\omega_2}$ . If now  $x - 1$  relays result in declaring  $\omega_1$  occurring before  $\omega_2$ , and  $|\mathcal{S}| - x + 1$  relays resulting in declaring otherwise, then server  $s$  replies with  $v_{\omega_1}$ . We fix  $|\mathcal{S}_1| = x - 1$  and  $|\mathcal{S}_2| = |\mathcal{S}| - x$  in  $\xi'_3$ .

Since no server receives relays from  $s_k$ , then every server receives  $x - 1$  relays (members of  $\mathcal{S}_1$ ) that declare that  $\omega_1$  occurs before  $\omega_2$ . Thus all servers reply with  $v_{\omega_2}$  to  $\rho_1$ . Hence  $\rho_1$  returns  $\omega_2$ . We extend  $\xi'_3$  by a second read operation  $\rho_2$  from  $r_2$  to obtain  $\xi''_3$ . Every server receives the request from  $\rho_2$  and relays  $\rho_2$ , and each server receives the relays from all other servers, this time including the relay of  $\rho_2$  from  $s_k$ . Since  $s_k$  follows *scheme2*, then every server observes that  $x - 1$  relays result in declaring that  $\omega_1$  occurs before  $\omega_2$ , and  $|\mathcal{S}| - x + 1$  relays result in the opposite. Thus all servers reply with  $v_{\omega_2}$  to  $\rho_2$  as well and hence  $\rho_2$  returns  $v_{\omega_2}$  in  $\xi''_3$ .

Consider finally an execution  $\xi_4$ , which is exactly the same as  $\xi''_3$ , with the difference that  $s_k$  follows *scheme1* instead of *scheme2*. Since no server receives relay messages from  $s_k$  during  $\rho_1$ , then all servers receive the same messages in  $\xi_4$  as in  $\xi''_3$  and they cannot distinguish the two executions. Thus, all servers observe  $x - 1$  relays that witness  $\omega_1$  before  $\omega_2$  and reply with  $v_{\omega_2}$  to  $\rho_1$ . As in  $\xi''_3$ ,  $\rho_1$  returns  $v_{\omega_2}$ . When

$\rho_2$  is performed, as before all servers receive the request of  $\rho_2$ , they relay  $\rho_2$  and each server receives the relays from all the other servers this time including the relay of  $\rho_2$  from  $s_k$ . However, since  $s_k$  follows *scheme1* then each server observes that  $x$  relays (the members of  $\mathcal{S}_1$  and  $s_k$ ) witnessed  $\omega_1$  before  $\omega_2$ . Thus, all servers reply according to our assumption with  $v_{\omega_1}$  to  $\rho_2$ , and  $\rho_2$  returns  $v_{\omega_1}$ . However, since  $\rho_1 \rightarrow \rho_2$ , read  $\rho_2$  violates atomicity.  $\square$

## 6 MWMR Algorithm Oh-MAM

Motivated by the impossibility result, we sought a solution that involves three or four communications exchanges per operation. We now present our MWMR algorithm Oh-MAM: *One and a half Round Multi-writer Atomic Memory*. We need to impose an ordering on the values that are concurrently written by multiple writers. Since writers may have the same local timestamp, we differentiate them by associating each value with a *tag* consisting of a pair of a timestamp *ts* and the *id* of the writer. We use lexicographic comparison to order tags (cf. [10]). The read protocol is identical to the SWMR setting, thus in Algorithm 2 we present only the pseudocode for writer and server processes.

---

### Algorithm 2 Writer and Server Protocols for MWMR algorithm Oh-MAM

---

<pre> 49: <i>At each writer</i> <math>w_i</math> 50: <b>Variables:</b> 51: <math>tag \in \langle \mathbb{N}^+, \mathcal{T} \rangle</math>, <math>v \in V</math>, <math>write\_op \in \mathbb{N}^+</math> 52: <math>maxTS \in \mathbb{N}^+</math>, <math>Q \subseteq \mathcal{S} \times M</math> 53: <b>Initialization:</b> 54: <math>tag \leftarrow \langle 0, w_i \rangle</math>, <math>v \leftarrow \perp</math>, <math>write\_op \leftarrow 0</math> 55: <math>maxTS \leftarrow 0</math>, <math>Q \leftarrow \emptyset</math> 56: <b>function</b> WRITE(<math>val</math> : input) 57:   <math>write\_op \leftarrow write\_op + 1</math> 58:   <b>broadcast</b> (<math>\langle discover, write\_op, w_i \rangle</math>) <math>\forall s \in \mathcal{S}</math> 59:   <b>wait until</b> <math> \mathcal{S} /2 + 1</math> <i>discoverAck</i> messages <math>m</math> 60:   with (<math>write\_op = m.write\_op</math>) 61:   Let <math>Q = \{(s, m)   r_i \text{ received message } m \text{ from } s\}</math> 62:   <math>maxTS \leftarrow \max\{m.tag.ts'   m \in Q\}</math> 63:   <math>\langle tag, v \rangle \leftarrow \langle \langle maxTS + 1, w_i \rangle, val \rangle</math> 64:   <math>write\_op \leftarrow write\_op + 1</math> 65:   <b>broadcast</b> (<math>\langle writeRequest, \langle tag, v \rangle, write\_op, w_i \rangle</math>) <math>\forall s \in \mathcal{S}</math> 66:   <b>wait until</b> <math> \mathcal{S} /2 + 1</math> <i>writeAck</i> messages <math>m</math> 67:   received with (<math>write\_op = m.write\_op</math>) 68:   <b>return</b> </pre>	<pre> 69: <i>At each server</i> <math>s_i</math> 70: <b>Variables:</b> 71: <math>tag \in \langle \mathbb{N}^+, \mathcal{T} \rangle</math>, <math>v \in V</math>, <math>write\_operations[1 \dots  \mathcal{W}  + 1]</math> 72: <b>Initialization:</b> 73: <math>tag \leftarrow \langle 0, s_i \rangle</math>, <math>v \leftarrow \perp</math>, <math>write\_operations[i] \leftarrow 0</math> for <math>i \in \mathcal{W}</math> 74: <b>Upon receive</b>(<math>\langle discover, write\_op, w_i \rangle</math>) 75:   <b>Send</b> (<math>\langle discoverAck, \langle tag, v \rangle, write\_op, s_i \rangle</math>) to process <math>w_i</math>. 76: <b>Upon receive</b>(<math>\langle writeRequest, tag', v', write\_op, w_i \rangle</math>) 77: <b>if</b> (<math>(tag &lt; tag') \wedge (write\_operation[w_i] &lt; write\_op)</math>) <b>then</b>. 78:   <math>\langle tag, v \rangle \leftarrow \langle tag', v' \rangle</math> 79:   <math>write\_operations[w_i] \leftarrow write\_op</math> 80: <b>send</b> (<math>\langle writeAck, \langle tag, v \rangle, write\_op, s_i \rangle</math>) to process <math>w_i</math> </pre>
--	--

---

**Writer Protocol.** This protocol is similar to [10]. When a write operation is invoked, a writer  $w$  broadcasts a *discover* message to all servers (line 58), and waits for  $|\mathcal{S}|/2 + 1$  *discoverAck* messages. Once the *discoverAck* messages are collected, writer  $w$  determines the maximum timestamp  $maxTS$  from the tags (line 62) and creates a new local tag  $tag$ , in which it assigns its id and sets the timestamp to one higher than the maximum,  $tag = \langle maxTS + 1, w_i \rangle$  (line 63). The writer then broadcasts a *writeRequest* message, including this tag, the value to be written, and its write operation,  $tag$ ,  $v$ , and  $write\_op$  to all servers (line 65). It then waits for  $|\mathcal{S}|/2 + 1$  *writeAck* messages (line 66) before terminating.

**Server Protocol.** Servers react to messages from the readers exactly as in Algorithm 1. Here we describe server actions for *discover* and *writeRequest* messages.

(1) Upon receiving message  $\langle discover, write\_op, w_i \rangle$ , server  $s$  attaches its local tag and local value in a new *discoverAck* message that it sends to writer  $w_i$ .

(2) Upon receiving a *writeRequest* message a server compares its local tag  $tag_s$  with the received tag  $tag'$ . In case where the message is not stale and server's local tag is older,  $tag_s < tag'$ , it updates its local timestamp and local value to those received (lines 77-79). Otherwise, no update takes place. Server  $s$  acknowledges this operation to writer  $w_i$  by sending it a *writeAck* message (line 80).

## 7 Correctness of MWMR Algorithm Oh-MAM

*Termination* of Algorithm 2 is satisfied with respect to our failure model as in Section 4. *Atomicity* is determined by lexicographically order the *tags* (instead of timestamps) in properties A1, A2, and A3 given in Section 4. Properties A1 and A3 can be proven as in Lemmas 3, 4, and 5. For paucity of space the complete proofs are in the optional Appendix. It is easy to see that the *tag* variable in each server  $s$  is monotonically increasing. This leads to the following lemma.

**Lemma 8** *In any execution  $\xi$  of the algorithm, if a server  $s$  receives a tag with value  $tag$  from process  $p$ , then  $s$  encloses a tag with value  $tag' \geq tag$  in any subsequent message.*

**Lemma 9** (Property A2) *In any execution  $\xi$  of the algorithm, if a write operation  $\omega_1$  writes a value with tag  $tag_1$  then for any succeeding write operation  $\omega_2$  that writes a value with tag  $tag_2$  we have  $tag_2 > tag_1$ .*

**Proof.** Let  $WSet_1$  be the set of servers that send a *writeAck* message within write operation  $\omega_1$ . Let  $Disc_2$  be the set of servers that send a *discoverAck* message within write operation  $\omega_2$ .

Based on the assumption, write operation  $\omega_1$  is complete. By Lemma 8, we know that if a server  $s$  receives a tag  $tag$  from a process  $p$ , then  $s$  includes tag  $tag'$  s.t.  $tag' \geq tag$  in any subsequently message. Thus the servers in  $WSet_1$  send a *writeAck* message within  $\omega_1$  with tag at least  $tag_1$ . Hence, every server  $s_x \in WSet$  obtains  $tag_{s_x} \geq tag_1$ .

When write operation  $\omega_2$  is invoked, it obtains the maximum tag,  $max\_tag$ , from the tags stored in at least a majority of servers. This is achieved by sending *discover* messages to all servers and collecting *discoverAck* replies from a majority of servers forming set  $Disc_2$  (lines 58-62 and 74-75).

Sets  $WSet_1$  and  $Disc_2$  contain a majority of servers, and so  $WSet_1 \cap Disc_2 \neq \emptyset$ . Thus, by Lemma 8, any server  $s_k \in WSet \cap Disc_2$  has a tag  $tag_{s_k}$  s.t.  $tag_{s_k} \geq tag_{s_x} \geq tag_1$ . Furthermore, the invoker of  $\omega_2$  discovers a  $max\_tag$  s.t.  $max\_tag \geq tag_{s_k} \geq tag_{s_x} \geq tag_1$ . The invoker updates its local tag by increasing the maximum tag it discovered, i.e.  $tag_2 = \langle max\_tag + 1, v \rangle$  (line 63), and associating  $tag_2$  with the value to be written. We know that,  $tag_2 > max\_tag \geq tag_1$ , hence  $local\_tag > tag_1$ .

Now the invoker of  $\omega_2$  includes its tag  $local\_tag$  with *writeRequest* message to all servers, and terminates upon receiving *writeAck* messages from a majority of servers. By Lemma 8,  $\omega_2$  receives *writeAck* messages with a tag  $tag_2$  s.t.  $tag_2 \geq local\_tag > tag_1$  hence  $tag_2 > tag_1$ , and the lemma follows.  $\square$

And the main result of this section follows.

**Theorem 10** *Algorithm Oh-MAM implements an atomic MWMR register.*

## 8 Implementation Efficiency

We now discuss the efficiency of our algorithms given in terms of latency of read and write operations, and the message complexity for each operation. We have earlier summarized these results in Table 1, where the complexity of Oh-SAM (Algorithm 1) and Oh-MAM (Algorithm 2) is compared to that of ABD [1].

Operation latency is measured in terms of the number of communication exchanges. Read operations in Algorithms 1 and 2 take 3 exchanges. Write operations in Algorithm 1 take 2 exchanges and in Algorithm 2 take 4 exchanges. Message complexity is measured in terms of the worst-case number of messages in each operation and it is obtained by examining the structure of communication exchanges. In Algorithms 1 and 2 the message complexity of read operations is  $|\mathcal{S}|^2 + 2|\mathcal{S}|$ . The message complexity of write operations in Algorithm 1 is  $2|\mathcal{S}|$  and in Algorithm 2 is  $4|\mathcal{S}|$ . Details are given in the optional Appendix.

## 9 Conclusions

In this work, we focused on the problem of emulating atomic read/write shared objects in message-passing settings using three communication exchanges. We presented a SWMR algorithm where each read operation completes in *three*, and each write operation in *two* communication *exchanges*. We showed that it is impossible to implement a MWMR atomic object when both read and write operations complete in *three* communication *exchanges*. Motivated by this, we presented the first MWMR algorithm where each read operation completes in *three*, and each write operation completes in *four* communication *exchanges*. We rigorously reasoned about the correctness of our algorithms. We note that the algorithms do not impose any constraints on the number of readers (SWMR and MWMR) or the writers (MWMR) participating in the service. Both algorithms are optimal in terms of *communication exchanges* when unbounded participation is assumed.

## References

- [1] ATTIYA, H., BAR-NOY, A., AND DOLEV, D. Sharing memory robustly in message passing systems. *Journal of the ACM* 42(1) (1996), 124–142.
- [2] DUTTA, P., GUERRAOU, R., LEVY, R. R., AND CHAKRABORTY, A. How fast can a distributed atomic read be? In *Proceedings of the 23rd ACM symposium on Principles of Distributed Computing (PODC)* (2004), pp. 236–245.
- [3] DUTTA, P., GUERRAOU, R., LEVY, R. R., AND VUKOLIĆ, M. Fast access to distributed atomic memory. *SIAM Journal on Computing* 39, 8 (2010), 3752–3783.
- [4] ENGLERT, B., GEORGIOU, C., MUSIAL, P. M., NICOLAOU, N., AND SHVARTSMAN, A. A. On the efficiency of atomic multi-reader, multi-writer distributed memory. In *Proceedings 13th International Conference On Principle Of Distributed Systems (OPODIS 09)* (2009), pp. 240–254.
- [5] GEORGIOU, C., NICOLAOU, N. C., AND SHVARTSMAN, A. A. On the robustness of (semi) fast quorum-based implementations of atomic shared memory. In *DISC '08: Proceedings of the 22nd international symposium on Distributed Computing* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 289–304.
- [6] GEORGIOU, C., NICOLAOU, N. C., AND SHVARTSMAN, A. A. Fault-tolerant semifast implementations of atomic read/write registers. *Journal of Parallel and Distributed Computing* 69, 1 (2009), 62–79.
- [7] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.
- [8] LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Trans. Comput.* 28, 9 (1979), 690–691.
- [9] LYNCH, N. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.
- [10] LYNCH, N. A., AND SHVARTSMAN, A. A. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of Symposium on Fault-Tolerant Computing* (1997), pp. 272–281.

## Appendix

### A Correctness: Oh-SAM for the SWMR setting

**Liveness.** Termination is satisfied with respect to our failure model: up to  $f$  servers may fail, where  $f < |\mathcal{S}|/2$  and each operation waits for messages from a majority of servers,  $|\mathcal{S}|/2 + 1$ . Let us consider this in more detail.

**Write Operation.** It is trivial to show that *liveness* is satisfied. Based on Algorithm 1, writer  $w$  creates a *writeRequest* message and then it broadcasts it to all the servers (line 24). Writer  $w$  then waits for *writeAck* messages from a majority of servers (lines 25-26). Based on our failure model, since we tolerate up to  $f < \frac{|\mathcal{S}|}{2}$  crashes, then writer  $w$  collects *writeAck* messages from a majority of live servers and write operation  $\omega$  terminates.

**Read Operation.** As a first step, reader  $r$  creates a *readRequest* message and it broadcasts it to all the servers (line 10). Since we tolerate up to  $f < \frac{|\mathcal{S}|}{2}$  crashes, then at least a majority of servers receives this *readRequest* message. Any server  $s$  that receives the *readRequest* message, broadcasts *readRelay* messages to all servers (lines 35-36). In addition, no server ever discards any incoming *readRelay* messages. Conversely, any server, whether it is aware or not of the *readRequest*, always keeps a record of the incoming *readRelay* messages and takes action as if it is aware of the *readRequest*. In the case where the *readRelay* messages were discarded because a server is not aware yet about a *readRequest*, then it may be the case that a server in the future may not be able to collect *readRelay* messages from a majority of servers. Hence, this would violate *liveness*. In more detail, the only difference between server  $s_i$  that received a *readRequest* message and server  $s_k$  that did not yet, is that  $s_i$  is able to broadcast *readRelay* messages, and  $s_k$  will broadcast *readRelay* messages when  $s_k$  receives the *readRequest* message (lines 35-36). Based on our failure model, each non-failed server will receive *readRelay* messages from a majority of servers and will send a *readAck* message to reader  $r$  (lines 47-48). Therefore, reader  $r$  will collect *readAck* messages from a majority of servers, decide on a value, and terminate (lines 14-16).

Based on the above, it will always be the case that acknowledgement messages, *readAck* and *writeAck*, from a majority of servers is collected in any read and write operation, hence *liveness* is satisfied.

The next lemma shows that the timestamp variable  $ts$  maintained by each server  $s$  in the system is monotonically increasing, allowing the ordering of the values according to their associated timestamps.

**Lemma 1.** *In any execution  $\xi$  of the algorithm, if a server  $s$  replies with a timestamp  $ts$  at time  $T$ , then server  $s$  replies with a timestamp  $ts' \geq ts$  at any time  $T' > T$ .*

**Proof.** The local timestamp  $ts$  of a server  $s$  is updated only when  $s$  receives *writeRequest* or *readRelay* messages. In both cases,  $s$  updates its local timestamp  $ts$  whenever it receives a higher timestamp (lines 38-39 and 42-43). Therefore, the server's local timestamp is monotonically increasing.  $\square$

**Lemma 2.** *In any execution  $\xi$  of the algorithm, if a server  $s$  receives a timestamp  $ts$  from process  $p$ , then  $s$  attaches a timestamp  $ts' \geq ts$  in any subsequent message it sends.*

**Proof.** When a server  $s$  receives a timestamp  $ts$  it updates its local timestamp  $ts_s$  only if  $ts > ts_s$  (lines 38-39 and 42-43). Since by Lemma 1 the local timestamp of the server monotonically increases, then at any subsequent time the local timestamp at  $s$  is  $ts_s \geq ts$ . Since  $s$  attaches a timestamp  $ts' = ts_s$  in any subsequent message, then  $ts' \geq ts$  and the lemma follows.  $\square$

At this point, we have to show that the timestamp is monotonically non-decreasing for the writer and the reader process.

**Lemma 11** *In any execution  $\xi$  of the algorithm, the variable  $ts$  stored in any process  $p$  is non-negative and monotonically non-decreasing.*



**Proof.** The proof is divided in two cases, for a *writer* and a *reader*.

*Writer case:* This is easy, as the writer increments its local timestamp  $ts$  every time it invokes a new write operation.

*Reader case:* Since Lemma 3 holds for any two read operations, then it also holds for the case where the two read operations are invoked from the same process  $p \in \mathcal{R}$ . Therefore, the timestamp of  $p$  is monotonically increasing and the lemma follows.  $\square$

## B Correctness: Oh-MAM for the MWMR setting

In order to show that Algorithm 2 is correct we have to show that the algorithm satisfies both *liveness* (termination) and *atomicity* (safety). Termination is satisfied with respect to our failure model: up to  $f$  servers may fail, where  $f < |\mathcal{S}|/2$  and each operation waits for messages from a majority of servers,  $|\mathcal{S}|/2 + 1$ . Let us examine this in more detail.

**Read Operation.** Since a read operation is the same as in Algorithm 1 for the SWMR setting, *liveness* is preserved as reasoned in section 4.

**Write Operation.** Writer  $w$  finds the maximum tag by broadcasting a *discover* message to all servers  $s \in \mathcal{S}$  and waiting to collect *discoverAck* replies from a majority of servers (lines 58-62 & 74-75). Since we tolerate up to  $f < \frac{|\mathcal{S}|}{2}$  crashes, then at least a majority of live servers will collect the *discover* message and reply to writer  $w$ . Once the maximum timestamp is found, then writer  $w$  updates its local tag (line 63) and broadcasts a *writeRequest* message to all servers  $s \in \mathcal{S}$ . Writer  $w$  then waits to collect *writeAck* replies from a majority of servers before it terminates. Again at least a majority of servers will collect the *writeRequest* message and will reply to writer  $w$ .

Based on the above, any read or write operation will collect a sufficient number of messages and terminate. Hence, the *liveness* is satisfied.

**Atomicity.** As given in Section 4, atomicity can be expressed in terms of *timestamps*. Since in the multiple writer-multiple reader (MWMR) setting we extend timestamps to tags, in this section we show how algorithm 2 satisfies *atomicity* using *tags* instead of *timestamps*.

*Monotonicity* allows the ordering of the values according to their associated tags. Thus, Lemma 12 shows that the tag  $tag$  variable maintained by each process  $p$  in the system is monotonically increasing.

**Lemma 12** *In any execution  $\xi$  of the algorithm, if a server  $s$  attaches a tag  $tag$  in a message at time  $T$ , then server  $s$  attaches a tag  $tag' \geq tag$  to a message at any time  $T' > T$ .*

**Proof.** The local tag  $tag$  of a server  $s$  is updated only when  $s$  receives either *readRelay* (lines 42-43) or *writeRequest* messages (lines 77-79). While receiving *readRequest* or *discover* messages respectively (lines 35-36 & 74-75), server  $s$  attaches its current timestamp without update. Therefore, server's local tag  $tag$  is monotonically increasing and the lemma follows.  $\square$

**Lemma 8.** *In any execution  $\xi$  of the algorithm, if a server  $s$  receives a tag with value  $tag$  from a process  $p$ , then  $s$  encloses a tag with value  $tag' \geq tag$  in any subsequent message.*

**Proof.** When server  $s$  receives a tag  $tag$  at time  $T$  it updates its local tag  $tag_s$  only if  $tag > tag_s$  (Algorithm 1 in lines 42-43 and Algorithm 2 in lines 77-79). Since by Lemma 12 the local tag of the server monotonically increases, then at time  $T' > T$ , the local tag at  $s$  is  $tag_s \geq tag$ . Since  $s$  attaches a timestamp  $tag' = tag_s$  in any message at time  $T'$ , then  $tag' \geq tag$  and the lemma follows.  $\square$

As a next step we show how atomicity property A3 is satisfied.

**Lemma 13 (Property A3)** *In any execution  $\xi$  of the algorithm, If  $\rho_1$  and  $\rho_2$  are two read operations such that  $\rho_1$  precedes  $\rho_2$ , i.e.,  $\rho_1 \rightarrow \rho_2$ , and  $\rho_1$  returns a tag  $tag_1$ , then  $\rho_2$  returns a tag  $tag_2 \geq tag_1$ .*

**Proof.** Let the two operations  $\rho_1$  and  $\rho_2$  be invoked by processes  $r_1$  and  $r_2$  respectively (not necessarily different). Let  $RSet_1$  and  $RSet_2$  be the sets of servers that reply to  $r_1$  and  $r_2$  during  $\rho_1$  and  $\rho_2$  respectively.

Let's suppose, for purposes of contradiction, that read operations  $\rho_1$  and  $\rho_2$  exist such that  $\rho_2$  succeeds  $\rho_1$ , i.e.,  $\rho_1 \rightarrow \rho_2$ , and the operation  $\rho_2$  returns a tag  $tag_2$  which is smaller than the  $tag_1$  returned by  $\rho_1$ , i.e.,  $tag_2 < tag_1$ .

According to our algorithm,  $\rho_2$  returns a tag  $tag_2$  which is smaller than the minimum tag received by  $\rho_1$ , i.e.,  $tag_1$ , if  $\rho_2$  discovers a pair  $\langle tag_2, v \rangle$  in the *readAck* message of some server  $s_x \in RSet_2$ , and  $tag_2$  is the minimum tag received by  $\rho_2$ .

Let us assume that server  $s_x$  replied with a pair  $\langle tag', v' \rangle$  to read operation  $\rho_1$ , i.e.,  $s_x \in RSet_1$ . By monotonicity of the timestamp at the servers (Lemma 12), and since  $\rho_1 \rightarrow \rho_2$ , then it must be the case that  $tag' \leq tag_2$ . Since, according to our assumption  $tag_1 > tag_2$ , and since  $tag_1$  is the smallest tag sent to  $\rho_1$  by any server in  $RSet_1$ , then it follows that  $r_1$  did not receive the *readAck* message from  $s_x$ , and hence  $s_x \notin RSet_1$ .

Now let us examine the actions of server  $s_x$ . From the algorithm, server  $s_x$  collects *readRelay* messages from a majority of servers in  $\mathcal{S}$  before sending *readAck* message to  $\rho_2$  (lines 49-50). Let  $RRSet_{s_x}$  denote the set of servers that sent *readRelays* to  $s_x$ . Since, both  $RRSet_{s_x}$  and  $RSet_1$  contain some majority of the servers then it follows that  $RRSet_{s_x} \cap RSet_1 \neq \emptyset$ .

This above means that there exists a server  $s_i \in RRSet_{s_x} \cap RSet_1$ , which sent (i) a *readAck* message to  $r_1$  for  $\rho_1$ , and (ii) a *readRelay* message to  $s_x$  during  $\rho_2$ . Note that  $s_i$  sends a *readRelay* message for  $\rho_2$  only after it receives a read request from  $\rho_2$  (lines 35-36). Since  $\rho_1 \rightarrow \rho_2$ , then it follows that  $s_i$  sent the *Read-Ack* message to  $\rho_1$  before sending the *readRelay* message to  $s_x$ . Thus, by Lemma 8, if  $s_i$  attaches a tag  $tag_{s_i}$  in the *readAck* to  $\rho_1$ , then  $s_i$  attaches a tag  $tag'_{s_i}$  in the *readRelay* message to  $s_x$ , such that  $tag'_{s_i} \geq tag_{s_i}$ . Since  $tag_1$  is the minimum tag received by  $\rho_1$ , then  $tag_{s_i} \geq tag_1$ , then  $tag'_{s_i} \geq tag_1$  as well. By Lemma 8, and since  $s_x$  receives the *readRelay* message from  $s_i$  before sending a *readAck* to  $\rho_2$ , it follows that  $s_x$  sends a tag  $tag_2 \geq tag'_{s_i}$ . Therefore,  $tag_2 \geq tag_1$  and this contradicts our initial assumption and completes our proof.  $\square$

At this point we have to show that any read operation that succeeds a write operation will receive *readAck* messages from the servers where each included a timestamp that is equal or greater than the one that the complete write operation returned.

**Lemma 14** *In any execution  $\xi$  of the algorithm, if a read operation  $\rho$  succeeds a write operation  $\omega$  that writes a pair  $\langle tag, v \rangle$ , i.e.,  $\omega \rightarrow \rho$ , and receives *readAck* messages from a majority of servers  $RSet$ , then each  $s \in RSet$  sends a *readAck* message to  $\rho$  with a tag  $tag_s$  s.t.  $tag_s \geq tag$ .*

**Proof.** Let us suppose, for the purposes of the proof, that  $WSet$  is the set of servers that send a *writeAck* message to the write operation  $\omega$  and let  $RRSet$  denote the set of servers that sent *readRelay* messages to server  $s$ .

Based on our assumption, write operation  $\omega$  is completed. By Lemma 8, we know that if server  $s$  receives a tag  $tag$  from process  $p$  at some time  $T$ , then  $s$  attaches a tag  $tag'$  s.t.  $tag' \geq tag$  in any message sent at any time  $T' \geq T$ . Thus a majority set of servers, forming  $WSet$ , sent a *writeAck* message to  $\omega$  with tag greater or equal to  $tag$ . Hence, every server  $s_x \in WSet$  has a tag  $tag_{s_x} \geq tag$ . Let us now examine a tag  $tag_s$  that server  $s$  sends to read operation  $\rho$ .

Before server  $s$  sends a *readAck* message to  $\rho$ , it must receive *readRelay* messages for the majority of servers,  $RRSet$  (lines 49-50). Since both  $WSet$  and  $RRSet$  contain a majority of servers, then it follows that  $WSet \cap RRSet \neq \emptyset$ . Thus, by Lemma 8, any server  $s_x \in WSet \cap RRSet$  has a tag  $tag_x$  s.t.  $tag_x \geq tag$ .

Since server  $s_x \in RSet$ , and by the algorithm server's  $s$  tag is always updated to the highest tag it observes (lines 42-43), then when server  $s$  receives the message from  $s_x$ , it will update its tag  $tag_s$  s.t.  $tag_s \geq tag_x$ .

Furthermore, server  $s$  creates a *readAck* message where it includes its local tag and its local value,  $\langle tag_s, v_s \rangle$ , and sends this *readAck* message within the read operation  $\rho$  (lines 49-50). Each  $s \in RSet$  sends a *readAck* to  $\rho$  with a tag  $tag_s$  s.t.  $tag_s \geq tag_x \geq tag$ . Therefore,  $tag_s \geq tag$  and the lemma follows.  $\square$

At this point we want to show that if a read operation succeeds a write operation, then it returns a value at least as recent as the one written.

**Lemma 15 (Property A1)** *In any execution  $\xi$  of the algorithm, if read operation  $\rho$  succeeds write operation  $\omega$  that writes  $\langle tag, v \rangle$ , i.e.,  $\omega \rightarrow \rho$ , and returns a timestamp  $tag'$ , then  $tag' \geq tag$ .*

**Proof.** Let's suppose that read operation  $\rho$  receives *readAck* messages from a majority of servers  $RSet$  and has to decide on a tag  $tag'$  associated with value  $v$  and then returns.

In this case, by Algorithm 1 (lines 14-16) it follows that read operation  $\rho$  decides on a tag  $tag'$  that belongs to a *readAck* message among the messages from servers in  $RSet$ ; and it will be the minimum tag among all the tags that are included in messages of servers  $RSet$ , hence  $tag' = min\_tag$ .

Furthermore, since  $tag' = min\_tag$  holds and from Lemma 14,  $min\_tag \geq tag$  holds, where  $tag$  is the tag returned from the last complete write operation  $\omega$ , then  $tag' = min\_tag \geq tag$  also holds. Therefore,  $tag' \geq tag$  holds and the lemma follows.  $\square$

## C Algorithms Complexity

In this section we assess the performance of our algorithms in terms of (i) latency of read and write operations, and (ii) the total number of exchanged messages in each read and write operation. The main two factors that affect the latency of each operation is (a) *computation time* and (b) *communication exchanges*.

**Computation Complexity.** This factor is measured by counting the steps that the algorithm takes each time it invokes an operation. In any operation at process  $p$ , it can receive up to  $|\mathcal{S}|$  messages. Since up to  $|\mathcal{S}|$  comparison steps can take place in each operation, the resulting computation complexity for each operation is  $|\mathcal{S}|$ .

**Communication Complexity.** Latency is measured in terms of communication exchanges. Below we analyse the communication complexity for each read/write operation for both algorithms.

*SWMR write operation communication complexity.* Based on Algorithm 1, writer  $w$  sends *writeRequest* messages to all servers  $s \in \mathcal{S}$  and waits for *writeAck* messages from a majority of servers. Once the *writeAck* messages are received, no further communication is required and the write operation terminates. Therefore, any write operation consists of 2 communication exchanges.

*MWMR write operation communication complexity.* Based on Algorithm 2, writer  $w$  sends *discover* messages to all servers  $s \in \mathcal{S}$  and waits for *discoverAck* messages from a majority of servers. Once the *discoverAck* messages are received, then writer  $w$  creates a new *writeRequest* message and propagates it to all servers  $s \in \mathcal{S}$ . It then waits for *writeAck* messages from a majority of servers. No further communication is required and the write operation terminates. Therefore, any write operation consists of 4 communication exchanges.

*Read operation (SWMR or MWMR) communication complexity.* As a first step, reader  $r$  sends *readRequest* message to all servers  $s \in \mathcal{S}$ , forming the first communication exchange. Once server  $s$  receives a *readRequest* message, it creates a *readRelay* message that it broadcasts to all servers  $s \in \mathcal{S}$ , forming the second communication exchange. Any active servers now wait to collect *readRelay* messages from a majority of servers. Once the *readRelay* messages are received, then the servers send a *readAck* message to the reader  $r$ , forming the third communication exchange. At this point, it is important to emphasize the fact that server  $s$  does not reply to any incoming *readRelay* messages. Therefore, any read operation consists of 3 communication exchanges in both the SWMR and MWMR settings.

**Message Complexity.** We measure message complexity as the number of exchanged messages in each read and write operation. For each read and write operation (in both SWMR and MWMR settings) we analyse message complexity in the *worst-case* scenario where failures do not exist, meaning that all messages from a sender process arrive to all destinations.

*SWMR write operation message complexity.* From the communication complexity analysis we know that a write operation of Algorithm 1 (SWMR) takes 2 communication exchanges before termination. The first exchange occurs when writer  $w$  sends a *writeRequest* message to all servers  $s \in \mathcal{S}$ . The second exchange occurs when all servers  $s \in \mathcal{S}$  send a *writeAck* message to writer  $w$ . Hence,  $2|\mathcal{S}|$  messages are exchanged in any write operation.

*MWMR write operation message complexity.* From the communication complexity analysis we know that a write operation in Algorithm 2 (MWMR) takes 4 communication exchanges to terminate. The first and the third exchange occurs when writer  $w$  sends *discover* and *writeRequest* messages to all servers  $s \in \mathcal{S}$  respectively. The second and fourth exchanges occur when servers  $s \in \mathcal{S}$  send *discoverAck* and *writeAck* messages back to writer  $w$ . Hence,  $4|\mathcal{S}|$  messages are exchanged in a single write operation.

*Read operation message complexity (SWMR and MWMR).* From the communication complexity analysis we know that a read operation takes 3 communication exchanges to terminate. The first exchange occurs when reader  $r$  sends a *readRequest* message to all servers  $s \in \mathcal{S}$ . The second exchange occurs when servers  $s \in \mathcal{S}$  send *readRelay* messages to all servers  $s \in \mathcal{S}$ . The final exchange occurs when servers  $s \in \mathcal{S}$  send a *readAck* message to reader  $r$ . Therefore,  $|\mathcal{S}|^2 + 2|\mathcal{S}|$  messages are exchanged in any read operation in both algorithms 1 (SWMR) and algorithm 2 (MWMR).